Indexed Matching Statistics and Shortest Unique Substrings

Djamal Belazzougui and Fabio Cunial

Helsinki Institute for Information Technology (HIIT)
Department of Computer Science, University of Helsinki, Finland*
name.surname@helsinki.fi

Abstract. The unidirectional and bidirectional matching statistics between two strings s and t on alphabet Σ, and the shortest unique substrings of a single string t, are the cornerstone of a number of large-scale genome analysis applications, and they encode nontrivial structural properties of s and t. In this paper we compute for the first time the matching statistics between s and t in $O((|s| + |t|) \log |\Sigma|)$ time and in $O(|s| \log |\Sigma|)$ bits of space, circumventing the need for computing the depths of suffix tree nodes that characterized previous approaches. Symmetrically, we compute for the first time the shortest unique substrings of a string t in $O(|t| \log |\Sigma|)$ time and in $O(|t| \log |\Sigma|)$ bits of space. A key component of our methods is an encoding of both the unidirectional and the bidirectional statistics that takes $2|t| + o(|t|)$ bits of space and that allows constant-time access to every position.

1 Introduction and Motivation

Let s and t be nonempty strings on alphabet $\Sigma = 1..\sigma$, let $\overline{s}$ and $\overline{t}$ be their reverse, and let $\$ = 0$ be a character not in Σ that is smaller than any character in Σ. In this paper we study the following concepts:

Definition 1. *Given two strings s and t and a threshold $\tau > 0$, the* unidirectional matching statistics $\mathsf{MS}_{t,s,\tau}$ *of t with respect to s is a vector of length $|t|$ that stores at index $i \in [0..|t| - 1]$ the length of the longest prefix of $t[i..|t| - 1]$ that occurs at least τ times in s.*

Definition 2. *Given a string t and a threshold $\tau > 0$, the* unidirectional distinguishing statistics $\mathsf{DS}_{t,\tau}$ *of t is a vector of length $|t|$ that stores at index $i \in [0..|t| - 1]$ the length of the shortest prefix of $t[i..|t| - 1]\$$ that occurs at most τ times in t.*

We drop any subscript from $\mathsf{MS}_{t,s,\tau}$ and $\mathsf{DS}_{t,\tau}$ whenever s, t or τ are clear from the context. Note that $\mathsf{DS}_{t,\tau}[i] = \mathsf{MS}_{t,t,\tau+1}[i] + 1$ for every i and τ.[1] $\mathsf{MS}_{t,s,1}$ has

* This work was partially supported by Academy of Finland under grant 250345 (Center of Excellence in Cancer Genetics Research).

[1] MS and DS have often been regarded as different problems in the literature: we thank an anonymous reviewer for making this connection explicit.

E. Moura and M. Crochemore (Eds.): SPIRE 2014, LNCS 8799, pp. 179–190, 2014.

also been called *external matching* [23], and $\mathsf{DS}_{t,1}$ has been called *distinguishing prefix* or *shortest unique substring* elsewhere [10]. By extension, $\mathsf{RS}_t = \mathsf{DS}_{t,1} - 1$ can be dubbed the *unidirectional repeating statistics* of t, since $\mathsf{RS}_t[i]$ is the length of the longest substring that starts at position i and that occurs at least twice in t. RS_t has also been called *internal matching* elsewhere [23]. $\mathsf{MS}_{t,s,1}$ and $\mathsf{DS}_{t,1}$ are almost as old as the suffix tree itself [23], with first applications to file transmission [22]. We are also interested in the bidirectional versions of such concepts:

Definition 3. *Given two strings s and t and a threshold τ, the* bidirectional matching statistics $\mathsf{BMS}_{t,s,\tau}$ *of t with respect to s is a vector of length $|t|$ that stores at index $i \in [0..|t|-1]$ the length of the longest substring $t[x..y]$ with $x \leq i \leq y$ that occurs at least τ times in s.*

Bidirectional distinguishing and repeating statistics, denoted respectively by $\mathsf{BDS}_{t,\tau}$ and $\mathsf{BRS}_{t,\tau}$, can be defined in the same way. Computing $\mathsf{MS}_{t,s,1}$ is a classical problem in string processing: the textbook solution scans t from left to right while navigating suffix links and child links in the suffix tree of s. Symmetrically, t can be scanned from right to left, while taking Weiner links and parent links in the (compressed) suffix tree of s [13]. Computing the depths of suffix tree nodes is the bottleneck of both approaches: such depths can be encoded either explicitly in $\Theta(|s| \log |s|)$ bits of space, and decoded in constant time [13], or implicitly on top of a compressed index, and decoded in $O(\log^{\epsilon} |s|)$ time [18]. In this paper we completely circumvent the need for computing the depths of suffix trees nodes, by indexing both s and $\overline{s}$ and by performing both a forward and a backward pass over t. This allows to compute $\mathsf{MS}_{t,s,\tau}$ in $O((|s|+|t|) \log \sigma)$ time and $O(|s| \log \sigma)$ bits of space for the first time, for any τ.

$\mathsf{DS}_{t,1}$ has been previously computed either in quadratic time using suffix trees [14], or in linear time using $O(|t| \log |t|)$ bits of space [10,20]. We adapt our MS algorithm to compute $\mathsf{DS}_{t,\tau}$ for the first time in $O(|t| \log \sigma)$ bits of space and in $O(|t| \log \sigma)$ time, for any τ.

A key component of our methods is an efficient encoding of $\mathsf{MS}_{t,s}$ and of DS_t, that takes $2|t| + o(|t|)$ bits of space and that allows to retrieve $\mathsf{MS}[i]$ and $\mathsf{DS}[i]$ in constant time for any i. This scheme uses ideas that have been previously applied to encode depths in compressed suffix trees [18]. Our index can represent $\mathsf{BMS}_{t,s}$ and BDS_t using just $o(|t|)$ bits of additional space, while still supporting constant-time access to the statistics at any position. Note that $\mathsf{BMS}_{t,s}$ has already been computed from $\mathsf{MS}_{t,s}$ in the past [19], but it has been encoded in $O(|t| \log |t|)$ bits of space.

Before proceeding, we note that fast and succinct representations of MS and DS enable a number of large-scale applications. For example, the profiles of MS and DS can be used to discriminate between sequencing errors and single-nucleotide variations in large read collections [15], and DS has applications in primer design for PCR, in comparative genomics [8], and in summarizing the context of the occurrences of a pattern in a large text collection [14]. More interestingly, MS and DS encode a number of structural properties of s and t. For example, recall that a *repeat* of t is any string w that occurs at least twice in t.

A repeat w is *maximal* if both awb and cwd occur in s, with $\{a,b,c,d\} \subseteq \Sigma$, $a \neq c$ and $b \neq d$. An occurrence i of a repeat w in t is said to be *exposed* if $t[i..i+|w|-1] = w$ and if no substring $t[i'..j']$ repeats in t, where $i' \leq i$, $j' \geq i+|w|-1$, and $(i',j') \neq (i, i+|w|-1)$. A *near-supermaximal repeat* of t is a repeat with at least one exposed occurrence. Clearly there is a near-supermaximal repeat exposed at position i in t if and only if $\mathsf{DS}_{t,1}[i] = \mathsf{DS}_{\overline{t},1}[|t|-i-\mathsf{DS}_{t,1}[i]+1]$, and its length is $\mathsf{DS}_{t,1}[i]-1$.

Symmetrically, recall that a *minimal absent word* of s is a string awb with $w \in \Sigma^*$ and $\{a,b\} \subseteq \Sigma$, such that both aw and wb occur in s, but awb does not occur in s. Clearly $t[j..j+\mathsf{MS}_{t,s,1}[j]]$ is a minimal absent word of s if and only if $\mathsf{MS}_{t,s,1}[j+1] \geq \mathsf{MS}_{t,s,1}[j]$, in which case $t[j+1..j+\mathsf{MS}_{t,s,1}[j]-1]$ is a maximal repeat of s.

Recall also that a *maximal exact match* (MEM) between s and t is a triple (i,j,ℓ) where $0 \leq i < |s|$, $0 \leq j < |t|$, and $1 \leq \ell \leq \min\{|s|,|t|\}$, such that $s[i..i+\ell-1] = t[j..j+\ell-1]$, $s[i-1] \neq t[j-1]$ and $s[i+\ell] \neq t[j+\ell]$ (we assume that $s[-1] \neq t[-1]$ and that $s[|s|] \neq t[|t|]$). A *maximal unique match* (MUM) between s and t is a MEM (i,j,ℓ) such that string $s[i..i+\ell-1]$ occurs exactly once in s and in t. MEMs, MUMs and their variants are used routinely in whole-genome alignment [11]. If $\mathsf{MS}_{t,s,1}[j] = \mathsf{MS}_{\overline{t},\overline{s},1}[|t|-j-\mathsf{MS}_{t,s,1}[j]]$, then there is a MEM $(i,j,\mathsf{MS}_{t,s,1}[j])$ at position j in t, and this is the longest MEM starting at j. This MEM is unique in t iff $\mathsf{MS}_{t,s,1}[j] \geq \mathsf{DS}_{t,1}[j]$, and it is unique in s iff $\mathsf{DS}_{t,s,1}[j]$ is defined, where $\mathsf{DS}_{t,s,\tau}$ is a binary version of distinguishing statistics. Note that $\mathsf{DS}_{t,s,1}$ can be implemented using $\mathsf{MS}_{t,s,2}$ and a bitvector $\mathtt{flag}$ such that $\mathtt{flag}[i] = 1$ iff $t[i..i+\mathsf{MS}_{t,s,1}-1]$ occurs exactly once in s: $\mathsf{DS}_{t,s,1}[i]$ is defined if and only if $\mathtt{flag}[i] = 1$, in which case $\mathsf{DS}_{t,s,1}[i] = \mathsf{MS}_{t,s,2}[i]+1$. We can thus decide in constant time whether a MUM starts at any position in t, and compute the length of such MUM.

Finally, given two positions $j > i$ in t, $\mathsf{MS}_{t,s,1}$ allows to compute the *average common substring* dissimilarity measure [21] between $t[i..j]$ and the whole s in $O(j-i)$ time, as well as the number k of factors in the relative LZ77 factorization of $t[i..j]$ with respect to s in $O(k)$ time. Besides having connections to Kolmogorov complexity and to the cross-entropy of finite-memory random sources [5], these measures are now the cornerstone of popular whole-genome comparison tools used in phylogenetics [21].

2 Computing and Indexing MS and DS

We denote by SA_s and BWT_s the suffix array and the Burrows-Wheeler transform of a string s, respectively. Recall that the *suffix array* $\mathsf{SA}_s[0,|s|]$ of s is the vector of indices such that $s[\mathsf{SA}_s[i],|s|]\$$ is the i-th smallest suffix of $s\$$ in lexicographical order, and the Burrows-Wheeler transform of s is the string $\mathsf{BWT}_s[0,|s|]$ satisfying $\mathsf{BWT}_s[i] = s[\mathsf{SA}_s[i]-1]$ if $\mathsf{SA}_s[i] > 0$, and $\mathsf{BWT}_s[i] = \$$ otherwise. We define the *suffix array range*, or identically the *BWT range* $(i_w,j_w)_s$ of a substring w in string s, as the maximal interval $[i_w..j_w]$ in SA_s such that all the suffixes $s[\mathsf{SA}_s[i]..|s|-1]$ for $i_w \leq i \leq j_w$ are prefixed by w.

We drop the subscript s from a range whenever the reference string is implied by the context. Incidentally, note that $\mathsf{RS}_t[i] = \max\{\mathsf{LCP}_t[j], \mathsf{LCP}_t[j+1]\}$, where $\mathsf{SA}[j] = i$ and LCP_t is the longest common prefix array of string t, i.e. $\mathsf{LCP}_t[j]$ is the longest prefix shared by suffixes $t[\mathsf{SA}_t[j]..|t|]\$$ and $t[\mathsf{SA}_t[j-1]..|t|]\$$ for all $j \in [1..|t|]$. Finally, we denote by $\neg s$ the complement of a bitstring s, i.e. $\neg s$ is the string t that satisfies $t[i] = 1 - s[i]$ for $0 \leq i < |s|$.

It is clear that $\mathsf{MS}_{t,s,\tau}[i] \geq 0$, $\mathsf{DS}_{t,\tau}[i] \geq 1$ and $\mathsf{RS}_t[i] \geq 0$ for all i and τ. The key additional property on which most of this paper rests is that $\mathsf{MS}_{t,s,\tau}$, $\mathsf{DS}_{t,\tau}$ and RS_t are *δ-monotone sequences* [17]:

Definition 4. *Let $a = a_0 a_1 \dots a_n$ and $\delta = \delta_1 \delta_2 \dots \delta_n$ be two sequences of non-negative integers. Sequence a is said to be* δ-monotone *if $a_i - a_{i-1} \geq -\delta_i$ for all $i \in [1..n]$.*

In particular, $\mathsf{MS}_{t,s,\tau}[i] - \mathsf{MS}_{t,s,\tau}[i-1] \geq -1$, $\mathsf{DS}_{t,\tau}[i] - \mathsf{DS}_{t,\tau}[i-1] \geq -1$ and $\mathsf{RS}_t[i] - \mathsf{RS}_t[i-1] \geq -1$ for all $i \in [1..|t|-1]$. Other popular examples of δ-monotone sequences in string processing are the *permuted LCP array* [18], and the *longest previous factor array* used in the LZ77 factorization of a string t, with $\delta_i = 1$ for all $i \in [1..|t|-1]$, and the lengths of the partial matches when searching a text t for a string w with the KMP algorithm[2].

It is natural to represent $\mathsf{MS}_{t,s,\tau}$ in succinct space by encoding the consecutive offsets $\mathsf{MS}_{t,s,\tau}[i] - \mathsf{MS}_{t,s,\tau}[i-1]$. It turns out that the same data structure can answer queries on $\mathsf{MS}_{\overline{t},\overline{s},\tau}[i]$ as well.

Lemma 1. *There is a data structure that takes $2|t| + o(|t|)$ bits of space and that answers queries on $\mathsf{MS}_{t,s,\tau}[i]$ and on $\mathsf{MS}_{\overline{t},\overline{s},\tau}[i]$ for any i in constant time.*

Proof. We follow the approach described in [18]. Specifically, we build a sequence $\mathtt{ms}_{t,s,\tau}$ of $2|t|$ bits by appending, for each $i \in [0, |t|-1]$ in increasing order, the binary string:

$$\underbrace{0\,0 \cdots\cdots\cdots 0}_{\substack{\mathsf{MS}_{t,s,\tau}[i] - \mathsf{MS}_{t,s,\tau}[i-1]+1 \\ \text{times}}}\ 1$$

where we set $\mathsf{MS}_{t,s,\tau}[-1] = 1$ for convenience. The resulting array contains either $2|t|$ or $2|t|-1$ bits: in the latter case, we append a final zero. Note that the number of zeros before the ith one in $\mathtt{ms}$ equals $i + \mathsf{MS}[i]$. Then, index $\mathtt{ms}$ to support $\mathtt{select}$ operations[3] in constant time using $2|t| + o(|t|)$ bits. We can thus compute $\mathsf{MS}[i]$ for any $i \in [0, |t|-1]$ by using the formula $\mathtt{select}(\mathtt{ms}, 1, i) - 2i$.

For a position $i \in [0, |t|-1]$, consider now the longest prefix of $\overline{t}[i..|t|-1]$ that occurs in $\overline{s}$, and assume that $i + \mathsf{MS}_{\overline{t},\overline{s},\tau}[i] = |t| - j$. Then substring $t[j..|t|-i-1]$ occurs in s, but substring $t[j-1..|t|-i-1]$ does not occur in s. Since the number of zeros before the $(j-1)$th one in $\mathtt{ms}$ is $j - 1 + \mathsf{MS}_{t,s,\tau}[j-1] < |t| - i$ and the

[2] Let i_x and i_{x+1} be the starting positions in t of two consecutive partial matches determined by KMP. In particular, let $t[i_x..i_x + a_x - 1] = w[0..a_x - 1]$ and $t[i_{x+1}..i_{x+1} + a_{x+1} - 1] = w[0..a_{x+1} - 1]$ for some positive maximal a_x and a_{x+1}. Then, $a_{x+1} - a_x \geq -\delta_{x+1}$, where δ_{x+1} is the shortest period of $w[0..a_x - 1]$.

[3] $\mathtt{select}(A, 1, i)$ is the position of the ith one in bitvector A, where i starts from zero.

number of zeros before the jth one in $\mathtt{ms}$ is $j + \mathsf{MS}_{t,s,\tau}[j] \geq |t| - i$, it follows that the ith zero from the right in $\mathtt{ms}$ is preceded by exactly $j = |t| - i - \mathsf{MS}_{\overline{t},\overline{s},\tau}[i]$ ones, therefore $\mathsf{MS}_{\overline{t},\overline{s},\tau}[i] = 2(|t| - i) - 1 - \mathtt{select}(\mathtt{ms}, 0, |t| - i - 1)$. □

Corollary 1. $\mathtt{ms}_{\overline{t},\overline{s},\tau} = \neg\overline{\mathtt{ms}_{t,s,\tau}}$

Proof. Since $\mathsf{MS}_{\overline{t},\overline{s},\tau}[i] = 2(|t| - i) - 1 - \mathtt{select}(\mathtt{ms}_{t,s,\tau}, 0, |t| - i - 1)$ and at the same time $\mathsf{MS}_{\overline{t},\overline{s},\tau}[i] = \mathtt{select}(\mathtt{ms}_{\overline{t},\overline{s},\tau}, 1, i) - 2i$, the position of the ith one from the left in $\mathtt{ms}_{\overline{t},\overline{s},\tau}$ equals the position of the ith zero from the right in $\mathtt{ms}_{t,s,\tau}$ for all i. □

We can encode $\mathsf{DS}_{t,\tau}$ in a similar bitvector $\mathtt{ds}_{t,\tau}$ with $2|t| + 1 + o(|t|)$ bits, by appending $\mathsf{DS}[i] - \mathsf{DS}[i-1] + 1$ zeros and a one for every $i \in [1..|t| - 1]$. We assume again $\mathsf{DS}[-1] = 1$, and we append a final zero if $\mathtt{ds}$ contains just $2|t|$ bits. As before $\mathsf{DS}[i] = \mathtt{select}(\mathtt{ds}, 1, i) - 2i$, but now $\mathsf{DS}_{\overline{t}}[i] = 2(|t| - i) + 1 - \mathtt{select}(\mathtt{ds}, 0, |t| - i)$. This implies that $2|t| + 1 - \mathtt{select}(\mathtt{ds}, 0, |t| - i) = \mathtt{select}(\mathtt{ds}_{\overline{t}}, 1, i)$, or in other words that $\mathtt{ds}_{\overline{t}} = 0 \cdot (\neg\overline{\mathtt{ds}_t[1..2|t|]})$.

More generally, the encoding described in Lemma 1 can be used to index any δ-monotone sequence $a_0 \ldots a_{n-1}$ in $n + a_{n-1} + \sum_{i=1}^{n-1} \delta_i$ bits, by concatenating $a_i - a_{i-1} + \delta_i$ zeros followed by a one for every i. The number of zeros before the ith one in this bitvector is $a_i + \sum_{j=1}^{i} \delta_j$, thus it is necessary to keep all the prefix sums of δ to answer queries on a_i.

We are interested in applications where the reference string s is fixed, and we have to output either $\mathtt{ds}_{s,\tau}$, or $\mathtt{ms}_{t,s,\tau}$ in reply to a query containing t. It turns out that both bitvectors can be derived from the Burrows-Wheeler transform of s and of $\bar{s}$, augmented with the corresponding suffix tree topologies.

Theorem 1. *Let* BWT_s *and* $\mathsf{BWT}_{\overline{s}}$ *be the Burrows-Wheeler transform of a string* s *and of its reverse* $\overline{s}$*, indexed to support a backward step in* α *time. Assume that we have a representation of the suffix tree topology of* s *and of* $\overline{s}$ *that supports parent operations in* β *time. Given a string* t *and a threshold* τ*, we can compute* $\mathtt{ms}_{t,s,\tau}$ *in* $O(|t|(\alpha + \beta))$ *time, and in* $O(\log|s| + \log|t|)$ *bits of space in addition to the input and the output.*

Proof. We apply twice the algorithm described in [13]. First, we scan t from right to left, using BWT_s and the suffix tree topology of s to determine the runs of consecutive ones in $\mathtt{ms}$. Specifically, we build a bitvector $\mathtt{runs}[1..|t| - 1]$ where $\mathtt{runs}[i] = 1$ iff $\mathsf{MS}[i] = \mathsf{MS}[i-1] - 1$, i.e. iff there is no zero between the ith and the $(i-1)$th ones in $\mathtt{ms}$. Assume that we have the interval $(i_w, j_w)_s$ in BWT_s that corresponds to substring $w = t[k..k + \mathsf{MS}[k] - 1]$ (the single-character interval for $k = |t| - 1$ can be directly derived from the table $C[1..\sigma]$ used in backward search). We try to perform a backward step using symbol $a = t[k-1]$: if the step leads to an interval of size at least τ, we set $\mathtt{runs}[k] = 1$ and we update the BWT interval to $(i_{aw}, j_{aw})_s$. Otherwise, we set $\mathtt{runs}[i] = 0$, we update the BWT interval to the interval of the parent of the proper locus of w in the suffix tree of s, and we try another backward step with character a. We repeat these operations until a backward step leads to an interval of size at least τ. Note that,

since we are not using the string depth operation, we don't know $\mathsf{MS}[k]$ for any $k < k^*$, where $k^* + \mathsf{MS}[k^*] = |t|$.

In the second phase we symmetrically scan t from left to right, using $\mathsf{BWT}_{\overline{s}}$, the suffix tree topology of $\overline{s}$, and vector $\texttt{runs}$, to build $\texttt{ms}$. Assume that we have the interval $(i_w, j_w)_{\overline{s}}$ in $\mathsf{BWT}_{\overline{s}}$ that corresponds to substring $w = t[k..h-1]$ such that $\mathsf{MS}[k] \geq h-k$ and $\mathsf{MS}[k-1] = h-k$ (again, we can derive the interval for $t[0]$ from the table $C[1..\sigma]$ used in backward search). We try to perform a backward step with symbol $t[h]$: if the step leads to an interval of size at least τ, we continue issuing backward steps with the following symbols of t, until we reach a position h^* in t such that a backward step with character $t[h^*]$ from the interval $(i_w, j_w)_{\overline{s}}$ of substring $w = t[k..h^*-1]$ leads to an interval of size less than τ. We thus know that $\mathsf{MS}[k] = h^* - k$, so we append $h^* - k - \mathsf{MS}[k-1] + 1 = h^* - h + 1$ zeros and a one to $\texttt{ms}$. Then, we iteratively replace $(i_w, j_w)_{\overline{s}}$ with the interval of the parent of the proper locus of w in the suffix tree of $\overline{s}$, and we try another backward step with symbol $t[h^*]$, until we reach an interval $(i_{w'}, j_{w'})_{\overline{s}}$ for which such backward step leads to an interval of size at least τ. Let this interval correspond to substring $w' = t[k'..h^*-1]$. Note that $\mathsf{MS}[k'] > \mathsf{MS}[k'-1] - 1$ and $\mathsf{MS}[x] = \mathsf{MS}[x-1] - 1$ for all $x \in [k+1..k'-1]$, therefore $\texttt{runs}[x]$ must be one for $x \in [k+1..k'-1]$ and $\texttt{runs}[k']$ must be zero, i.e. k' is the index of the first zero to the right of position k in $\texttt{runs}$. We can thus append $k' - k - 1$ ones to $\texttt{ms}$ and repeat the process from substring $t[k'..h^*]$ and its interval in $\mathsf{BWT}_{\overline{s}}$.

If we store vector $\texttt{runs}$ in the last $|t|-1$ bits of $\texttt{ms}$, each iteration of the second phase of the algorithm overwrites only parts of $\texttt{runs}$ that will not be used in following iterations. This is easy to see, and is left to the reader. □

The two-pass approach of Theorem 1 completely avoids string-depth operations. Recall that computing the depth of a suffix tree node ultimately requires to decompress a position in the underlying compressed suffix array: if such array is encoded in $O(|s| \log \sigma)$ bits, the best known time complexity for this operation is $O(\log^{\epsilon} |s|)$. Complexity increases when the compressed suffix array is encoded in $|s| \log \sigma + o(|s|)$ bits or in $|s| \log \sigma(1 + o(1))$ bits. Note also that the algorithm in Theorem 1 uses either BWT_s or $\mathsf{BWT}_{\overline{s}}$ at every step, i.e. it does not need to keep their intervals synchronized. A similar result holds for $\texttt{ds}_{s,\tau}$:

Theorem 2. *Let BWT_s and $\mathsf{BWT}_{\overline{s}}$ be the Burrows-Wheeler transform of a string s and of its reverse $\overline{s}$, indexed to support a backward step in α time. Assume that we have a representation of the suffix tree topology of s and of $\overline{s}$ that supports parent operations in β time. We can compute $\texttt{ds}_{s,\tau}$ in $O(|s|(\alpha+\beta))$ time, and in $O(\log |s|)$ bits of space in addition to the input and the output.*

Proof. By applying almost verbatim the two-phase approach of Theorem 1. In the first phase we build vector $\texttt{runs}$ by trying a backward step with character $a = s[k-1]$ from the interval in BWT_s of the longest string that starts at position k and that occurs more than τ times, i.e. from the interval of string $w = s[k..k + \mathsf{DS}_{s,\tau}[k] - 2]$. If this step leads to an interval of size greater than τ, we set $\texttt{runs}[k] = 1$ and we repeat from the BWT interval of aw. Otherwise, we set $\texttt{runs}[k] = 0$, we move to the interval of the parent of the proper locus of w in the suffix tree of s, and we

try another backward step with character a. We repeat these operations until a backward step leads to an interval of size at most τ.

In the second phase, assume again that we know the interval in $\mathsf{BWT}_{\overline{s}}$ of the longest string that starts at position k and that occurs more than τ times, i.e. the interval of string $w = s[k..h-1]$ with $h = k + \mathsf{DS}_{t,\tau}[k] - 1$. We iteratively move to the interval of the parent of the proper locus of w in the suffix tree of $\overline{s}$ and we try a backward step with character $a = s[h]$, until such a step leads to an interval of size greater than τ. Thanks to `runs`, we know that the interval from which the last backward step was taken corresponds to string $s[k'..h-1]$, where k' is the position of the first zero to the right of k in `runs`. We can thus append $k' - k - 1$ ones to `ds` and move to string $s[k'..h]$. We then try backward steps with the characters at position $h+1, h+2, \ldots$ until a backward step reaches an interval of size at most τ: this gives $\mathsf{DS}[k']$. □

Corollary 2. *Given a string s, there is a data structure that allows to compute: (1) $\mathtt{ds}_{s,\tau}$ in $O(|s| \log \sigma)$ time and in $O(\log |s|)$ bits of space in addition to the input and the output; (2) $\mathtt{ms}_{t,s,\tau}$ for any string t, in $O(|t| \log \sigma)$ time and in $O(\log |s| + \log |t|)$ bits of space in addition to the input and the output. This data structure takes $2|s| \log \sigma + O(|s|)$ bits of space and can be built in $O(|s| \log \sigma)$ time using $O(|s| \log \sigma)$ bits of space.*

Proof. BWT_s and $\mathsf{BWT}_{\overline{s}}$ can be built in $O(|s| \log \log \sigma)$ time and $O(|s| \log \sigma)$ bits of space using the algorithm described in [9]. The wavelet trees on BWT_s and $\mathsf{BWT}_{\overline{s}}$ can then be built in $O(|s| \log \sigma)$ time. The suffix tree topologies of s and $\overline{s}$ can be built in $O(|s| \log \sigma)$ time using just the corresponding wavelet trees, using the approach described in [1,2]. □

Corollary 3. *Given a string s, there is a data structure that allows to compute: (1) $\mathtt{ds}_{s,\tau}$ in $O(|s|)$ time and in $O(\log |s|)$ bits of space in addition to the input and the output; (2) $\mathtt{ms}_{t,s,\tau}$ for any string t, in $O(|t|)$ time and in $O(\log |s| + \log |t|)$ bits of space in addition to the input and the output. This data structure takes $2|s| \log \sigma + o(|s| \log \sigma)$ bits of space and can be built in randomized $O(|s|)$ time using $O(|s| \log \sigma)$ bits of space.*

Proof. Given a bitvector of length n, we can build in $O(n)$ time a data structure that supports constant-time select queries and that takes $2n + o(n)$ bits of space [4,12]. We achieve the claimed time complexity by plugging in Theorem 1 the index described in [3], which supports constant-time backward steps and takes $2|s| \log \sigma + o(|s| \log \sigma)$ bits of space. The latter is built in randomized $O(|s|)$ time and $O(|s| \log \sigma)$ bits of space, using the algorithm described in [1,2]. The suffix tree topologies are then built in $O(|s|)$ time from the indexes of [3]. □

3 Computing and Indexing BMS and BDS

We start by generalizing the algorithm for computing BMS from MS described in [19], in order to make it work on user-defined *blocks*, rather than on positions,

of the input string. We say that a pair (i, j) is a *matching statistics interval* of string t if $i \in [0..|t| - 1]$ and $j = i + \mathsf{MS}_{t,s,\tau}[i] - 1$. We say that an interval is *maximal* if it is not contained into any other interval, and we denote the list of all maximal intervals by $\mathcal{I}$. The following properties are immediate, and will be used extensively in the sequel.

Property 1. There is at most one interval in $\mathcal{I}$ that ends at any given position j in t. Given an interval (i, j), the maximal interval that contains (i, j) is $(j - \mathsf{MS}_{\overline{t},\overline{s},\tau}[|t| - j - 1] + 1, j)$.

Property 2. Let $(i_0, j_0), (i_1, j_1), \ldots, (i_m, j_m)$ be a subset of $\mathcal{I}$ sorted by increasing starting position. Then $\sum_{h=1}^{m} i_h - i_{h-1} \leq |t|$ and $\sum_{h=1}^{m} j_h - j_{h-1} \leq |t|$.

It is natural to compute $\mathsf{BMS}_{t,s,\tau}$ by scanning t and $\mathsf{MS}_{t,s,\tau}$ while keeping in memory the *active* subset of $\mathcal{I}$, i.e. the set of intervals in $\mathcal{I}$ that cover the current position in t. The following algorithm is an alternative to [19]:

Lemma 2 ([19]). *$\mathsf{BMS}_{t,s,\tau}$ can be computed from $\mathsf{MS}_{t,s,\tau}$ in $O(|t|)$ time and $O(|t| \log |t|)$ bits of space.*

Proof. Assume that we are at position k in t, and let I be the subset of $\mathcal{I}$ that contains all the intervals that start before position k and that cover position k, i.e. all intervals $(i, j) \in \mathcal{I}$ with $j = i + \mathsf{MS}[i] - 1$ and $i < k \leq j$. We implement I as a doubly-linked list, with a node for every distinct length of an interval in I. The node associated with length ℓ stores all the intervals of length ℓ in I as a doubly-linked list. We assume that I is sorted by decreasing length, and we keep it sorted during the algorithm using insertion sort. Moreover, we use array $\texttt{ends}[0..|t| - 1]$ to store in $\texttt{ends}[j]$ a pointer to the only interval in I that ends at j, if any. Let $\texttt{prev}$ be a pointer to the *previous interval* $(i_{prev}, j_{prev}) = (k-1, k-2+\mathsf{MS}[k-1])$ if it belongs to I ($\texttt{prev}$ is null if such interval does not belong to I), and let $\texttt{last}$ be a pointer to the node of I that contains the interval $(i, j) \in I$ with maximum i. Let $\delta_k = \mathsf{MS}[k] - \mathsf{MS}[k-1]$.

If $\delta_k = -1$, then $\mathsf{BMS}[k]$ is the length of the first node in I, list I does not change, and we set $\texttt{prev}$ to null. Similarly, if $\mathsf{MS}[k] = 1$, then $\mathsf{BMS}[k]$ is the length of the first node in I, or one if I is empty, list I does not change, and we set $\texttt{prev}$ to null. Otherwise, we proceed as follows. If $\texttt{prev} \neq \emptyset$, we remove from I the interval pointed by $\texttt{prev}$ and we set $\texttt{ends}[j_{prev}]$ to null. Then, we add to I the current interval $(k, k + \mathsf{MS}[k] - 1)$, as follows.

Let ℓ be the length of node $\texttt{last}$. If $\mathsf{MS}[k] = \ell$ we add the new interval to $\texttt{last}$, otherwise we scan I linearly starting from the node associated with length ℓ, until we find the node associated with length $\mathsf{MS}[k]$ (we create a new node if no such node exists). This linear scan visits at most $|\mathsf{MS}[k] - \ell| = |j_h - j_{h-1} - i_h + i_{h-1}|$ nodes of I for every maximal interval $(i_h, j_h) \in \mathcal{I}$, thus it visits in total $\sum_{h=1}^{m} |j_h - j_{h-1} - i_h + i_{h-1}| \leq \sum_{h=1}^{m} j_h - j_{h-1} + i_h - i_{h-1} \leq 2|t|$ nodes of I. Note that this corresponds to charging the moves along I to the bits of $\texttt{ms}$.

After having added $(k, k+\mathsf{MS}[k]-1)$ to I, we update $\texttt{prev}$, $\texttt{last}$ and $\texttt{ends}[k+\mathsf{MS}[k] - 1]$ to point to the newly inserted interval. Once again, $\mathsf{BMS}[k]$ equals

the length of the first node in I. Before moving to position $k+1$, we remove from I the interval pointed by $\texttt{ends}[k]$, if any. □

Lemma 3. *$\mathsf{BDS}_{t,\tau}$ can be computed from $\mathsf{DS}_{t,\tau}$ in $O(|t|)$ time and $O(|t|\log|t|)$ bits of space.*

Proof. By applying the algorithm in Lemma 2 almost verbatim. Now, for every position k in t, I stores all the *minimal intervals* that start before k and cover k, i.e. all the intervals that start before k, cover k, and do not contain any other interval. I is sorted by increasing length. We insert interval $(k, k+\mathsf{DS}[k]-1)$ in I for every position k in t, but if $\mathsf{DS}[k]-\mathsf{DS}[k-1]=-1$ we first remove the interval (i_{prev}, j_{prev}) pointed by $\texttt{prev}$, since it is not minimal, and we set $\texttt{ends}[j_{prev}]$ to null. We thus have the guarantee that there is at most one interval in I that ends at every position j of t. It is easy to see that the moves along I can still be charged to the bits of $\texttt{ds}$. □

Note that the notion of interval holds for any δ-monotone sequence $a = a_0 a_1 \ldots a_n$, and the algorithms in Lemma 2 and 3 allow to compute, respectively, the length of a longest and of a shortest interval of a that covers every position $i \in [0..n]$ in the sequence, in time linear *in the size of the index of the sequence*. In particular, the algorithm in Lemma 2 can be applied to compute BRS_t, the bidirectional version of the repeating statistics, from the unidirectional RS_t.

Assume now that string t is the concatenation of m nonempty substrings, i.e. that $t = t_0 \cdot t_1 \cdots t_{m-1}$ with $t_i \in \Sigma^+$ for all $i \in [0..m-1]$. We call such substrings *blocks* in what follows, and we denote with $\beta(j)$ the block that contains position j in t. We assume that block boundaries are marked in a bitvector of size $|t|$, indexed to support rank operations[4]. We say that an interval *spans* block i if it starts before block i and if it ends after block i. The algorithm described in Lemma 2 can be adapted to compute the length of a longest matching statistics interval (or, symmetrically, of a shortest distinguishing statistics interval) that spans every block $i \in [1..m-2]$.

Lemma 4. *The length of a longest matching statistics interval that starts before block i and ends after block i, for every $i \in [1..m-2]$, can be computed from $\mathsf{MS}_{t,s,\tau}$ in $O(|t|)$ time.*

Proof. We apply again the algorithm in Lemma 2 almost verbatim. Now array $\texttt{ends}$ has one position per block, and $\texttt{ends}[k]$ points to the longest interval in I that starts before block k and ends inside block k.

Assume that we are processing block k. Let $r = \max\{\beta(i+\mathsf{MS}[i]-1) : \beta(i) = k\}$ be the rightmost block that contains the ending position of an interval that starts inside block k. Similarly, let $\ell = \max\{\mathsf{MS}[i] : \beta(i) = k\}$ be the length of a longest interval that starts inside block k. We call any interval that starts inside block k and that ends inside block r a *farthest interval*, and any interval

[4] $\texttt{rank}(A, 1, i)$ is the number of ones in bitvector A up to position i, included.

of maximum length ℓ a *longest interval*[5]. Let (i_r, j_r) be a farthest interval of maximum possible length, i.e. $j_r - i_r = \max\{\mathsf{MS}[i] : \beta(i) = k, \beta(i + \mathsf{MS}[i] - 1) = r\}$. This interval spans every block $h \in [k+1..r-1]$, and no interval that starts inside block k and spans the same blocks is longer, thus (i_r, j_r) will be active in the following iterations. Similarly, if $j_r - i_r < \ell$, let (i_ℓ, j_ℓ) be a longest interval. Clearly no interval that starts inside block k and that spans the same blocks as (i_ℓ, j_ℓ) is longer, thus (i_ℓ, j_ℓ) will be active in the following iterations.

When we process block k, we first remove from I the interval that ends inside block k, by following the pointer in $\texttt{ends}[k]$, and we assign to block k the length of the first node in I. Then, we insert in I the interval (i_ℓ, j_ℓ) (if it exists), and then the interval (i_r, j_r). If pointer $\texttt{ends}[\beta(\texttt{j}_\ell)]$ is null, we store in $\texttt{ends}[\beta(\texttt{j}_\ell)]$ a pointer to (i_ℓ, j_ℓ). Otherwise, if the length of interval $\texttt{ends}[\beta(\texttt{j}_\ell)]$ is smaller than ℓ, we remove interval $\texttt{ends}[\beta(\texttt{j}_\ell)]$ from I and we store in $\texttt{ends}[\beta(\texttt{j}_\ell)]$ a pointer to (i_ℓ, j_ℓ). This is because the old interval spans the same blocks as the new interval, but it is shorter. We do the same for (i_r, j_r).

Insertions in I work in the same way as before, with `last` pointing to the interval $(i, j) \in I$ with maximum i. The total number of movements in I can be still bounded by $2|t|$, since we are inserting a subset of $\mathcal{I}$. □

Lemma 5. *The length of a shortest distinguishing statistics interval that starts before block i and ends after block i, for every $i \in [1..m-2]$, can be computed from $\mathsf{DS}_{t,\tau}$ in $O(|t|)$ time.*

Proof. By adapting the algorithm in Lemma 4. Assume that we are at block k, and let L be the list of all tuples $(i, j, j - i, \beta(j))$ where $j = i + \mathsf{DS}[i] - 1$, and $\beta(i) = k$. We sort L by increasing third component of each tuple (length), and then we stable-sort L by the last component of each tuple (ending block). Then, we scan the sorted L and we insert in I the first interval that we find associated with every block h, updating the corresponding pointers in `ends` if the previous interval is longer than the new interval. The total number of movements in I after insertions can be still bounded by $2|t|$, since we are inserting a subset of the minimal intervals we inserted in Lemma 3. □

Once again, these blocked variants of BMS and BDS can be applied to any δ-monotone sequence (thus in particular to BRS) in time linear in the size of the index for such sequence.

As done with MS and DS, we would like to build succinct indexes that support *bidirectional* queries in constant time. To this end, we augment `ms` and `ds` by exploiting the following property, whose immediate proof is left to the reader:

Property 3. Given a position $j \in [0..|t|-1]$, the position $i^* = \min\{i \in [0..j-1] : j \leq i + \mathsf{MS}[i] - 1\}$ can be computed by $\texttt{select}(\texttt{ms}, 0, j) - j + 1$.

Note that the same property holds for DS, for RS, and for every δ-monotone sequence $a_0 a_1 \ldots a_n$ with $\delta_i = 1$ for all $i \in [1..n]$. However, the property does not generalize to δ-monotone sequences where δ_i is not constantly one for all $i \in [1..n]$.

[5] If all blocks have the same length, a longest interval spans every block in $[k+1..r-2]$.

Indeed, in such cases position j is covered by position $i < j$ iff the ith one in the index of the sequence is preceded by at least $j + 1 + \sum_{k=1}^{i}(\delta_k - 1)$ zeros.

Theorem 3. *There is a data structure that takes $2|t| + o(|t|)$ bits of space and that answers queries on $\mathsf{BMS}_{t,s,\tau}[i]$ for any i in constant time.*

Proof. We store `ms` in $2|t|$ bits, and we augment it with an index that takes $O(|t| \log\log|t| / \log|t|) \in o(|t|)$ bits, and that supports rank and select queries in constant time [7,16]. We assume that we can read any block of $\Theta(\log|t|)$ consecutive bits of `ms` in constant time. Moreover, we partition `ms` into $B = \lceil 2|t|/\log|t| \rceil$ blocks of size $\log|t|$ each. We use again the notation $\beta(i)$ to identify the block that contains position i in `ms`. For every block k, let (i_k, j_k) be a longest matching statistics interval of t such that $\beta(\texttt{select}(\texttt{ms}, 1, i_k)) = k$. We store the position of $\texttt{select}(\texttt{ms}, 1, i_k)$ inside each block in array $\texttt{start}[0..B-1]$, using $2|t| \log\log|t| / \log|t|$ bits. Then, we build a range-maximum data structure RMQ on the pairs $(k, j_k - i_k)$ for $k \in [0..B-1]$, using $2(2|t|/\log|t|) + o(|t|) = 4|t|/\log|t| + o(|t|/\log|t|)$ bits of space [6].

Given a position j in t, we use Property 3 to compute i^*, the smallest $i < j$ with $j \leq i + \mathsf{MS}[i] - 1$. Let $p = \beta\big(\texttt{select}(\texttt{ms}, 1, i^*)\big)$ and $q = \beta\big(\texttt{select}(\texttt{ms}, 1, j)\big)$. Since all the matching statistics intervals that cover position j must start between i^* and j, we query RMQ with the pair $(p+1, q-1)$ to get the block $k \in [p+1..q-1]$ with longest interval in constant time, and we compute the length ℓ_1 of such interval by $\ell_1 = h - 2 \cdot \texttt{rank}(\texttt{ms}, 1, h)$ where $h = k \log|t| + \texttt{start}[k]$. Finally, we load in constant time blocks p and q and we use the Four Russians technique to compute in constant time the lengths ℓ_2 and ℓ_3 of the longest matching statistics intervals that starts inside block p and q, respectively, using a precomputed table of size $o(|t|)$ bits. We finally return $\max\{\ell_1, \ell_2, \ell_3\}$. □

Corollary 4. *There is a data structure that takes $2|t| + o(|t|)$ bits of space and that answers queries on $\mathsf{BDS}_t[i]$ for any i in constant time.*

Proof. By applying the approach in Theorem 3 verbatim, using `ds` instead of `ms` and a range-minimum rather than a range-maximum data structure. □

References

1. Belazzougui, D.: Linear time construction of compressed text indices in compact space. In: Proceedings of the 46th ACM Symposium on Theory of Computing. ACM (2014)
2. Belazzougui, D.: Linear time construction of compressed text indices in compact space. ArXiv preprint ArXiv:1401.0936 (2014)
3. Belazzougui, D., Navarro, G.: Alphabet-independent compressed text indexing. ACM Transactions on Algorithms 10(4) (2014)
4. Clark, D.: Compact Pat Trees. PhD thesis, University of Waterloo, Canada (1996)
5. Farach, M., Noordewier, M., Savari, S., Shepp, L., Wyner, A., Ziv, J.: On the entropy of DNA: Algorithms and measurements based on memory and rapid convergence. In: Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 48–57 (1995)

6. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM Journal on Computing 40(2), 465–492 (2011)
7. Golynski, A.: Optimal lower bounds for rank and select indexes. Theoretical Computer Science 387(3), 348–359 (2007)
8. Haubold, B., Pierstorff, N., Möller, F., Wiehe, T.: Genome comparison without alignment using shortest unique substrings. BMC Bioinformatics 6(1), 123 (2005)
9. Hon, W.-K., Sadakane, K., Sung, W.-K.: Breaking a time-and-space barrier in constructing full-text indices. SIAM J. Comput. 38(6), 2162–2178 (2009)
10. İleri, A.M., Külekci, M.O., Xu, B.: Shortest unique substring query revisited. In: Kulikov, A.S., Kuznetsov, S.O., Pevzner, P. (eds.) CPM 2014. LNCS, vol. 8486, pp. 172–181. Springer, Heidelberg (2014)
11. Kurtz, S., Phillippy, A., Delcher, A.L., Smoot, M., Shumway, M., Antonescu, C., Salzberg, S.L.: Versatile and open software for comparing large genomes. Genome Biology 5(2), R12 (2004)
12. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
13. Ohlebusch, E., Gog, S., Kügel, A.: Computing matching statistics and maximal exact matches on compressed full-text indexes. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 347–358. Springer, Heidelberg (2010)
14. Pei, J., Wu, W.-H., Yeh, M.-Y.: On shortest unique substring queries. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp. 937–948. IEEE (2013)
15. Philippe, N., Salson, M., Commes, T., Rivals, E.: CRAC: An integrated approach to the analysis of RNA-seq reads. Genome Biology 14(3), R30 (2013)
16. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms (TALG) 3(4), 43 (2007)
17. Robertson, M.M.: A generalization of quasi-monotone sequences. Proceedings of the Edinburgh Mathematical Society (Series 2) 16(01), 37–41 (1968)
18. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems 41(4), 589–607 (2007)
19. Schnattinger, T., Ohlebusch, E., Gog, S.: Bidirectional search in a string with wavelet trees and bidirectional matching statistics. Inf. Comput. 213, 13–22 (2012)
20. Tsuruta, K., Inenaga, S., Bannai, H., Takeda, M.: Shortest unique substrings queries in optimal time. In: Geffert, V., Preneel, B., Rovan, B., Štuller, J., Tjoa, A.M. (eds.) SOFSEM 2014. LNCS, vol. 8327, pp. 503–513. Springer, Heidelberg (2014)
21. Ulitsky, I., Burstein, D., Tuller, T., Chor, B.: The average common substring approach to phylogenomic reconstruction. Journal of Computational Biology 13(2), 336–350 (2006)
22. Weiner, P.: The file transmission problem. In: Proceedings of the National Computer Conference and Exposition, June 4-8, pp. 453–453. ACM (1973)
23. Weiner, P.: Linear pattern matching algorithms. In: Switching and Automata Theory, pp. 1–11. IEEE (1973)